

Microcontroller Basics Course (5)

part 5: the UART, timers and interrupts

By B. Kainka

The previous instalments of the Microcontroller Basics Course have primarily concentrated on programming languages. Now it's time to take a closer look at the microcontroller hardware. This instalment deals with the UART, timers and interrupts.

The internal serial interface of every 8051 microcontroller makes it possible to create a simple data link to a PC. Since the interface is implemented as an autonomous hardware UART, the processor is scarcely burdened by this task. Bytes are received and sent by the UART operating fully in the background. The only things requiring attention are correct initialisation and the ongoing transfer of data to and from the UART.

To understand exactly how the UART is programmed, we need to look at the special function registers (SFR) for the serial interface. To initialise the interface, we must first load SCON register with suitable parameters (see **Table 1**).

SBUF (SFR 099h) is the data register of the serial interface. It is actually a 'front' for two registers, namely the transmit data register and the receive data register. A transmit process is initiated by simply performing a write access to the SBUF register. In the other direction, a received byte can be read from SBUF. In 9-bit mode, the ninth bit (TB8 or RB8) can always be found in the SCON register.

Usually only operating mode 1 is used, which means an 8-bit UART with the baud rate generated by Timer 1. The 8-bit data stream is initiated by one start bit and terminated by one stop bit. The baud rate is 1/16 (if SMOD = 1) or 1/32 (if SMOD = 0) of the overflow rate of Timer 1. SMOD is the most significant bit of the PCON register (Power Control, SFR 87h), which is otherwise responsible for controlling the power-down modes

Table 1 SCON (SFR 98h)

7	6	5	4	3	2	1	0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0 SM1 Serial Mode bits 1 & 2 select the operating mode:

0 0 Mode 0: 8-bit shift register

0 1 Mode 1: 8-bit UART, baud rate via Timer 1

1 0 Mode 2: 9-bit UART, 375 kBaud at 12 MHz

1 1 Mode 3: 9-bit UART, baud rate via Timer 1

SM2: multiprocessor mode

REN: Receiver Enable

TB8: Transmitted Bit 8 for 9-bit mode

RB8: Received Bit 8 for 9-bit mode

TI: Transmitter Interrupt, '1' if transmission successful

RI: Receiver Interrupt, '1' following receipt of a character

(see **Table 2**).

Generating a clock signal for the serial interface requires first programming Timer 1. Each of the two timers in the 8051 has two 8-bit counter registers that can be loaded and read. The 89S8252, by the way,

also has a third timer, but we will disregard it for now.

TL0 (SFR 8Ah): Timer 0, low byte

TH0 (SFR 8Ch): Timer 0, high byte

TL1 (SFR 8Bh): Timer 1, low byte

TH1 (SFR 8Dh): Timer 1, high byte

Table 2 PCON (SFR 87h)

7	6	5	4	3	2	1	0
SMOD	—	—	—	GF1	GF2	PD	IDL

SMOD 1 = high baud rate, 0 = low baud rate

GF1, GF2 freely usable flags

PD Power-Down mode (only for CMOS)

IDL Idle mode

Table 3 TMOD (SFR 89h)

7	6	5	4	3	2	1	0
Gate	C/T	M1	M0	Gate	C/T	M1	M0
Counter 1				Counter 0			
Gate				The associated counter is enabled via the Int0 or Int1 pin.			
C/T				0: Timer, 1: Counter			
M1 M0				Operating mode			
0 0				13-bit timer/counter			
0 1				16-bit timer/counter			
1 0				8-bit timer/counter with automatic reloading			
1 1				only for Counter 0: two separate 8-bit counters			

Table 4 TCON (SFR 88h)

7	6	5	4	3	2	1	0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
TF1: overflow flag for Timer 1							
TR1: start Timer 1							
TF0: overflow flag for Timer 0							
TR0: start Timer 0							
IE1: interrupt flag for Int1							
IT1: interrupt on Int1 with edge triggering							
IE0: interrupt flag for Int0							
IT0: interrupt on Int0 with edge triggering							

Listing 1. The serial interface in assembler.

```

;Serial port access          (COMPORT.ASM)
;11,059MHz, 9600 Baud
#include 8051.H
        .org 0000H

INIT    clr    TR1            ;stop timer 1
        mov    TH1,#0FAH      ;256-6: 9600 baud
        mov    TL1,#0FAH
        anl    TMOD,#0FH      ;Timer1: 8 bit auto-reload
        orl    TMOD,#20H
        setb   TR1            ;start timer
        mov    SCON,#50H      ;InitRS232
        setb   TI
        orl    PCON,#80H      ;SMOD=1

NEXT    acall  RX
        mov    P1,A           ;Port write
        nop
        nop
        mov    A,P1           ;Port read
        acall  TX
        sjmp   NEXT

RX      jnb    RI,RX
        mov    A,SBUF
        clr    RI
        ret

TX      jnb    TI,TX
        clr    TI
        mov    SBUF,A
        ret
.end

```

The properties of the timers are controlled via registers TCON (SFR 88h) and TMOD (SFR 89h), while the operating mode is configured using TMOD (see Table 3).

Timer 1 should be used as a free-running counter counting the internal clock pulses of the 8051. It is thus configured as a mode-2 timer. Each time the counter overflows, a pre-set counter state is reloaded into the counter.

Each counter is enabled via its TR bit in the TCON register. TCON also contains the overflow flags for both timers as well as four control bits for interrupt-driven control via the INT0 and INT1 pins (see Table 4).

In order to use Timer 1 as a baud rate generator, all we have to do is to set the TR1 bit. This bit is bit-addressable and can be accessed at bit address 8Eh. However, first the division factors must be configured with both timers stopped. In operating mode 2, the high counter byte (TH1) contains the reload value, which is reloaded into the actual counter register (TL1) each time an overflow occurs. For a transmission rate of 9600 baud, a clock frequency of $(16 \times 9.6 \text{ kHz}) = 153.6 \text{ kHz}$ is required if the SMOD bit is set. Since the counter is clocked by the internal clock, which runs at 1/12 of the crystal frequency, a division factor of 6 should be set if the crystal frequency is 11.0592 MHz:

$$11059.2 \text{ kHz} \div 12 \div 6 \div 16 = 9.6 \text{ kHz}$$

A division factor of 6 is achieved by using a reload value of $(256 - 6) = 250$ (FAh), since the counter counts up.

Serial data transmission

The UART can be used with any desired programming language. Using it with assembler is a particularly good way to illustrate the involved processes. **Listing 1** shows a sample program in which the interface is used to provide access to Port 1 via RS232. A PC can thereby modify and read the port status.

For initialisation, all relevant registers are first loaded with their control parameters. The main routine NEXT then needs only the two subroutines TX and RX. The receive subroutine RX first polls the RI bit until it finds that a character is present. The receive buffer SBUF is then read and RI is reset. In the other direction, the transmit subroutine TX polls the TI bit until the previously transmitted byte has been fully processed. The byte to be sent is then written to the send buffer SBUF, which starts the transmit portion of the UART.

Each received byte is transferred to Port 1 in the main routine. After a short delay generated by two NOP instructions, the program

reads the status of Port 1 and sends back the byte it has read via the serial interface. We have thus implemented a parallel-to-serial converter that can be used only for port outputs, only for read accesses to the port or for mixed operation. All leads to be used as inputs must be set high by the sent data in order to put them in a high-impedance state. The effect of sending a value of 255 (FFh), for example, is to cause the entire port to be available as an input port and initiate a read transaction.

This program can be tested using the program Terminal.exe, for example, which was already used in the December 2001 issue of *Elektor Electronics* in connection with an IR transceiver. The significant difference here is that what is being transferred is bytes rather than numerical values in text format. Figure 1 shows an example of how the port can be driven. The first two control bytes are returned unchanged, which means that the microcontroller reports that the sent port state was present at P1 with no changes. The third byte (255) sets all port pins High, which makes them usable as inputs. This time, however, the response from the microcontroller shows that one line is being pulled to ground externally. This program can thus be used in this manner to only poll all eight port lines.

It's even easier to use the serial interface in C than in assembler, since suitable modules are already present. The serial interface can be initialised by `InitSerialPort0` using the parameter `DEF_SIO_MODE`. The functions `getc()` and `putc()` transfer individual bytes. A program having exactly the same functions as the previously described assembler program can thus be easily written (see **Listing 2**).

In BASIC-52, the UART is already initialised by the system. It is no problem to do the same job in Basic. However, in this case it is easier to work with numerical values in text format, since the Basic interpreter always keeps an eye on the data received via the serial interface (it's always possible for a Ctrl-C to appear if the user wants to stop a program). The Input command thus always receives whole lines, which must be terminated with CR. **Listing 3** shows a very simple program for the 'remote control' of Port 1. These functions can be tried out directly using the Basic terminal program.

Interrupts

In many cases it is necessary for the microcontroller to perform several tasks quasi-concurrently. An effective way to realise this is to use interrupt handling, with a main routine that does its own work most of the time but

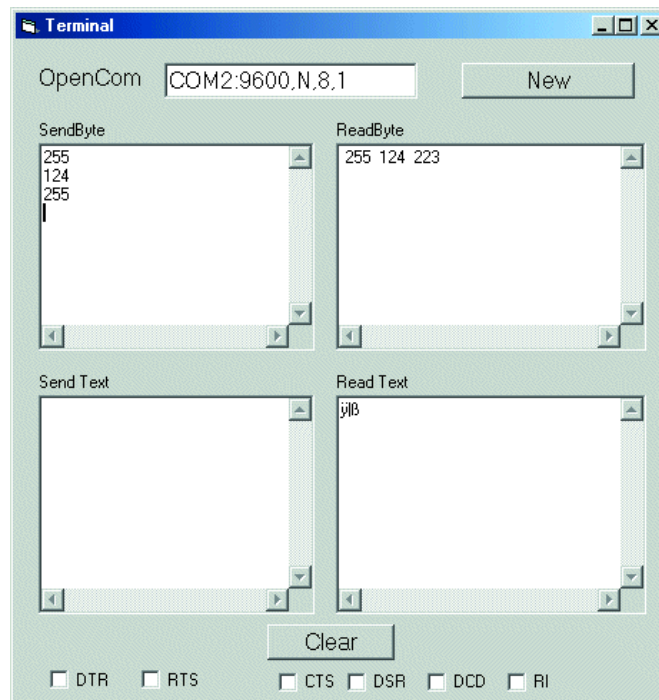


Figure 1. Port outputs and port polling.

it can be briefly interrupted by special events in order to perform a second task.

The principle of such an interruption can be illustrated very easily using BASIC-52. Every 8051 microcontroller has two interrupt inputs called `Int0` (P3.2) and `Int1` (P3.3). BASIC-52 supports `Int1` with the

`ONEX1` command. Whenever the P3.3 line is pulled Low externally, this Low edge triggers an interrupt. The program then branches to the line number given after `ONEX1`. The actual interrupt subroutine must end with `RETI`. **Listing 4** shows a simple example. Here a counting loop with incrementing port outputs runs in

Listing 2. The UART in READS-51.

```
// ----- READS51 generated header -----
// module : uart.c
// -----

#define TRUE 1
#define FALSE 0

#include <sfr51.h>
#include <Sio51.h>

main(){
char n;

// -- initialize serial port (9600 Baud) --
InitSerialPort0(DEF_SIO_MODE);
//DEF_SIO_MODE is defined in <Sio51.h>

// endless loop
while(TRUE)
{
n=getc();
P1=n;          // Port output
n=P1;          // Port read
putc(n);
}
}
```

Listing 3. Port accesses in BASIC-52

```

1  REM Port I/O (PORTIO.BAS)
10 INPUT N
20 PORT1=N
30 N=PORT1
40 PRINT N
50 GOTO 10

```

Listing 4. Interrupt control in BASIC-52.

```

10  REM Interrupt (INT1.BAS)
100 ONEX1 500
200 REM main prog
210 FOR N=0 TO 255
220 PORT1=N
230 NEXT N
240 GOTO 210
500 REM Interrupt subroutine
510 PRINT "Interrupt P3.3"
520 RETI

```

the foreground and the subroutine called by the interrupt sends a message to the PC.

BASIC-52 also has the ONTIME command, which reacts to a particular time event in precisely the same manner. However, these are only a few of many possible sources of interrupts. The processor also recognises interrupts from its timers and the serial interface. Several interrupts can also be employed in a single program, in which case it is necessary to specify the priorities assigned to the individual interrupts. The interrupt with the highest priority is allowed to interrupt all other interrupt routines.

The processor has a central control element for enabling interrupts in the form of the Interrupt Enable (IE) register (see **Table 5**). The EA bit disables or enables all of the configured interrupts.

For each interrupt there is an associated interrupt address to which the processor automatically jumps. This address must hold a

jump instruction to the starting address of the corresponding interrupt routine, which in turn must end with a reti instruction.

0023h	SINT	Serial interface
001Bh	TIMER1	Timer 1
0013h	EXTI1	External interrupt 1 (P3.3)
000Bh	TIMER0	Timer 0
0003h	EXTI0	External interrupt 0 (P3.2)
0000h	RESET	Reset

The next example (**Listing 5**) shows the serial interface being used with interrupts. In our first serial interface example, the program remains 'stuck' in the receive routine each time it is waiting for a new character to arrive, but here an interrupt is triggered when a character is received.

Received characters are directly transferred to register R7. A square-wave generator whose period is set by the value located in R7 runs in the

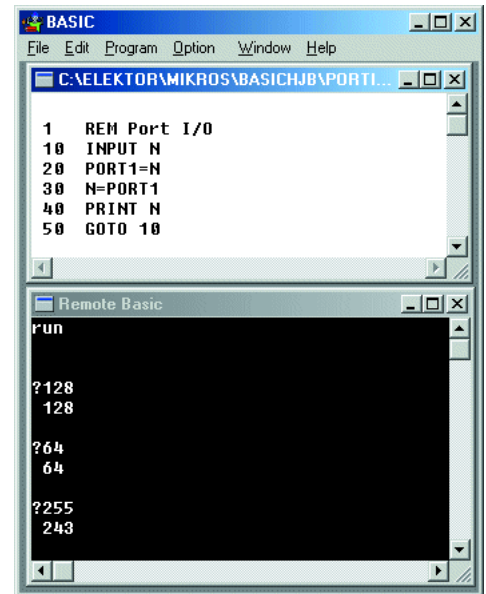


Figure 2. Direct access to Port 1.

main routine. This means that the frequency of the generator can be externally changed without any noticeable response-time delay.

Now we want to show you how to handle interrupts in READS-51. This time Timer 0 should trigger an interrupt. Here you have to keep in mind that Timer 1 is often already used for the serial interface, so it is not available for time control. The 89S8252 also has a third timer, Timer 2, which is used in BASIC-51 (for instance) to generate the baud rate. Timer 2 is very versatile, but you must remember that it is not present in the smaller 8051 derivatives and may also have different properties in certain microcontrollers (such as the 80535). If we use Timer 0 here for time control, we can thus use our program with all other 8051-compatible microcontrollers.

Here we want to attempt to generate a precise square-wave signal with an edge spacing of 1 ms (see **Listing 6**). Timer 0 and Timer 1 each receive a clock signal at one twelfth of the crystal frequency. If the crystal frequency is 11.059 MHz, we have to count to 921 to obtain an interval of 1 ms. Since we are dealing with up counters that trigger interrupts when they overflow, we must specify a value of $(65536 - 921) = 64516$ (FC67 hex). This value is loaded into the counter registers TH0 and TL0 the first time the counter is started and then again each time an interrupt is triggered by a carry-out signal.

An interrupt function can be created very easily in READS-51 using the keyword 'interrupt' and the microcontroller's associated interrupt address. It is also necessary to initialise the timer in the Main function, and the appropriate interrupt bits must be set.

Table 5 IE (SFR A8h)

7	6	5	4	3	2	1	0
EA	—	—	ES	ET1	EX1	ET0	EX0

EA	enable all interrupts
ES	enable serial interrupts for sending and receiving
ET1	enable Timer 1 interrupt via TF1
EX1	enable external interrupt 1 (P3.3)
ET0	enable Timer 0 interrupt via TF0
EX0	enable external interrupt 0 (P3.2)

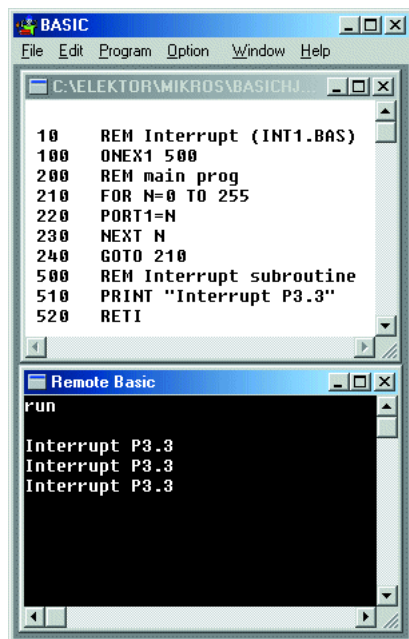


Figure 3. Interrupt-driven message generation.

This program is suitable for investigating the execution times of C programs. An oscilloscope connected to port P1.0 will measure a precisely symmetrical square-wave signal. The duration of each level is approximately 1.25 ms. The code generated by the compiler thus requires around 250 μ s for the actual jump to the interrupt function and the re-initialisation of the timer. If a more accurate 1-ms timing event is needed, you can try fine-tuning the initial value of the counter.

The timer has four different modes that can be used to cover various time ranges. In this example we use mode 1, which can generate timing events of up to 65 ms. Without reloading inside the interrupt function, the following times can be achieved (in round numbers):

- Mode 0: 13-bit timer (8 ms)
- Mode 1: 16-bit timer (65 ms)
- Mode 2: 8-bit auto-reload (0–0.25 ms)
- Mode 3: TH0 = 8-bit timer (0.25 ms)

In these examples, we have seen how the same techniques can be used in different programming languages. Suitable initialisation of the relevant function registers is a prerequisite for using the hardware. With a bit of experience, you will be able to glean the appropriate settings from an existing program and then use them with a different programming language.

(010208-6)

Listing 5. Interrupt-driven character reception.

```
;Serial Interrupt      (COMINT.ASM)
;11,059 MHz, 9600 Baud
#include 8051.H
        .org 0000H
        ljmp INIT
        .org 0023H
        ljmp RX

INIT     clr     TR1           ;stop timer 1
        mov     TH1,#0FAH     ;256-6: 9600 baud
        mov     TL1,#0FAH
        anl     TMOD,#0FH     ;Timer1: 8 bit auto-reload
        orl     TMOD,#20H
        setb    TR1           ;start timer
        mov     SCON,#50H     ;InitRS232
        setb    TI
        orl     PCON,#80H     ;SMOD=1
        mov     IE,#90H      ;EA +ES

NEXT     mov     A,R7
        mov     R1,A
        mov     A,#255
        mov     P1,A
ON        djnz   R1,ON
        mov     A,R7
        mov     R1,A
        mov     A,#0
        mov     P1,A
OFF       djnz   R1,OFF
        sjmp    NEXT

RX        mov     R7,SBUF
        clr     RI
        reti

        .end
```

The next (and final) instalment of this course deals with the LC display, which is

mapped into the address space of the external memory as a set of components.

Listing 6. A timer interrupt in C.

```
// ----- READS51 generated header -----
// module : C:\Rigel\Reads51\Work\Inter\Inter.c
// created : 09:16:45, Thursday, March 07, 2002
// -----
#include <sfr51.h>

void interrupt (0x000B) square(void)
{
    P1_0 = ! P1_0;
    TR0=0;           //stop timer 0
    TH0=0xFC;        //timer 0 1ms
    TL0=0x67;
    TR0=1;           //start timer 0
}

main(){
    int n;
    n=0;
    TH0=0xFC;        //timer 0 1ms
    TL0=0x67;
    TMOD=TMOD & 0xF0;
    TMOD=TMOD | 0x01; //timer 0 16 bit
    TR0=1;           //start timer 0
    ET0=1;           //timer 0 interrupt
    EA=1;            //enable interrupts
    while(1);
}
```